# PRÁCTICAS DE SISTEMAS OPERATIVOS II

## PRIMERA PRÁCTICA EVALUABLE

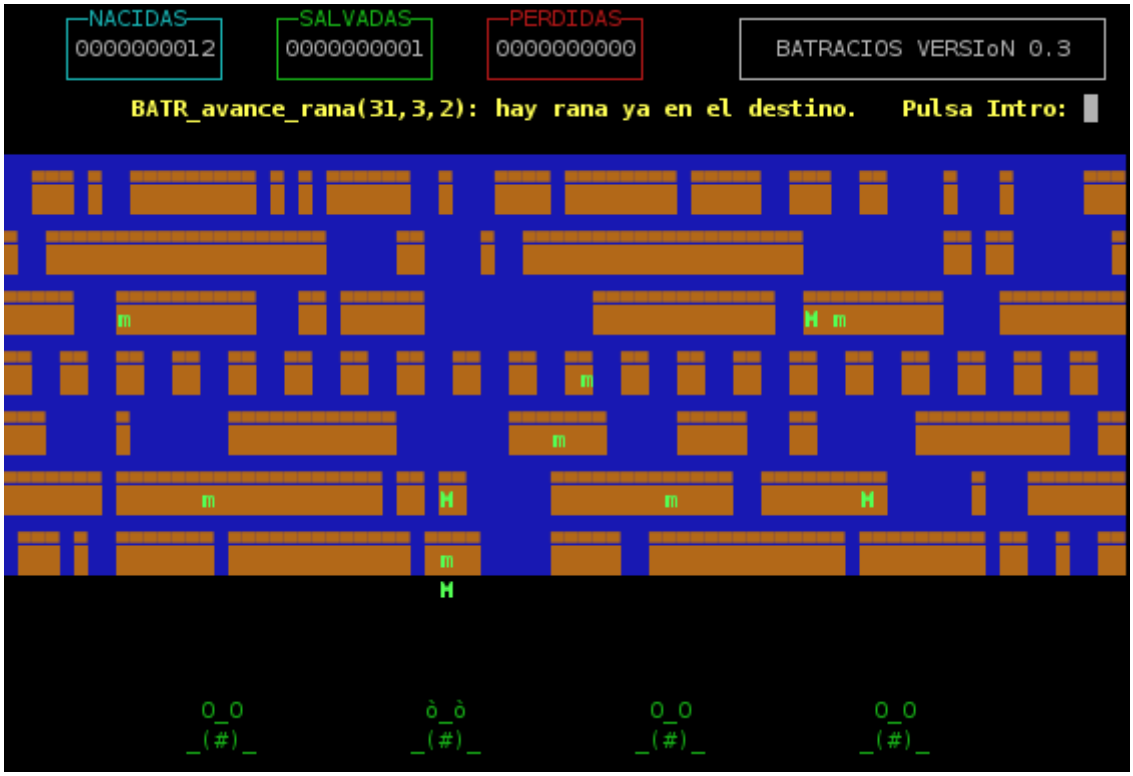
## Batracios

---

1. **Enunciado.**

   En la práctica que vais a realizar podréis aprender a usar los nuevos mecanismos IPC recientemente aprendidos. Se trata de simular mediante un programa la vida de unas ranas, inspirada en el famoso juego de consola clásico "Frogger".

   Según se va ejecutando el programa, se ha de ver una imagen parecida a la siguiente:




   En la imagen pueden observarse las ranas, representadas por una "m" de color verde. Las ranas nacen de una de cuatro ranas madre de color verde oscuro y situadas en la parte inferior de la pantalla. El objetivo de sus vidas es atravesar un río. Para ello, pueden moverse hacia adelante, hacia la derecha o hacia la izquierda según su criterio, pero nunca hacia atrás. Deberán mantenerse dentro de la pantalla. Dos ranas no pueden ocupar la misma posición.

   Cuando llegan las ranas a la orilla inferior del río, deben tener cuidado. Solo pueden saltar a las posiciones donde se encuentran unos troncos flotando sobre su superficie. Si saltan sobre el agua, se produce un error. Para aumentar la dificultad, los troncos están a la deriva moviéndose continuamente. Una rana puede desaparecer debido a que el tronco sobre el que se encuentra sale de la pantalla. Eso no constituye un error. Esa rana se ha "perdido".

   En la parte superior de la pantalla aparecen tres contadores. El primero cuenta las ranas que han nacido. El segundo, las ranas que han alcanzado la orilla superior del río y se han puesto a salvo y el tercero, lleva cuenta de aquellas que se han perdido porque su tronco desapareció de la pantalla. Al

final de la práctica, el número de ranas nacidas debe coincidir con el de ranas salvadas más el de ranas perdidas más el de ranas que permanecen en la pantalla.

El tiempo de la práctica se mide en "tics" de reloj. La equivalencia entre un tic y el tiempo real es configurable. Puede ser incluso cero. En ese caso la práctica irá a la máxima velocidad que le permita el ordenador donde se esté ejecutando.

En esta práctica usaréis una biblioteca de enlazado estático que se os proporcionará. El objetivo es doble: por un lado aprender a usar una de tales bibliotecas y por otro descargar parte de la rutina de programación de la práctica para que os podáis centrar en los problemas que de verdad importan en esta asignatura.

El programa constará de un único fichero fuente, `batracios.c`, cuya adecuada compilación producirá el ejecutable `batracios`. Respetad las mayúsculas/minúsculas de los nombres.

Para simplificar la realización de la práctica, se os proporciona una biblioteca estática de funciones (`libbatracios.a`) que debéis enlazar con vuestro módulo objeto para generar el ejecutable. Gracias a ella, algunas de las funciones necesarias para realizar la práctica no las tendréis que programar sino que bastará nada más con incluir la biblioteca cuando compiléis el programa. La línea de compilación del programa podría ser:

```
gcc batracios.c libbatracios.a -o batracios -lm
```

Disponéis, además, de un fichero de cabeceras, `batracios.h`, donde se encuentran definidas, entre otras cosas, las macros que usa la biblioteca y las cabeceras de las funciones que ofrece.

El proceso inicial se encargará de preparar todas las variables y recursos IPC de la aplicación y registrar manejadoras para las señales que necesite. Este proceso, además, debe tomar e interpretar los argumentos de la línea de órdenes y llamar a la función `BATR_inicio` con los parámetros adecuados. El proceso será responsable de crear los procesos adicionales necesarios, salvo los procesos que representan a las ranitas, que son hijos del proceso que maneja a la rana madre correspondiente. En ningún caso, puede la práctica mantener en ejecución simultánea más de 25 procesos. Las cuatro ranas madres procreadoras y cada una de la pequeñas ranitas que nazcan, serán representadas, por lo tanto, mediante un proceso. También es responsabilidad del primer proceso el controlar que, si se pulsa CTRL+C, la práctica acaba, no dejando procesos en ejecución ni recursos IPCs sin borrar. La práctica devolverá 0 en caso de ejecución satisfactoria o un número mayor que cero, en caso de detectarse un error.

La práctica se invocará especificando dos parámetros obligatorios desde la línea de órdenes. Si no se introducen argumentos, se imprimirá un mensaje con la forma de uso del programa por el canal de error estándar. El primer argumento será un número entero comprendido entre 0 y 1000 que indicará la equivalencia en ms de tiempo real de un tic de reloj. O dicho de otro modo, la "lentitud" con que funcionará la práctica. El segundo argumento es la media de tics de reloj que necesita una rana madre descansar entre dos partos. Es un número entero estrictamente mayor que 0. Si el primer parámetro es 1 o mayor, la práctica funcionará tanto más lenta cuanto mayor sea el parámetro y no deberá consumir CPU apreciablemente. El modo de conseguir la rapidez o lentitud lo realiza la propia biblioteca. Vosotros no tenéis más que pasar dicho argumento a la función de inicio. Si es 0, irá a la máxima velocidad, aunque el consumo de CPU sí será mayor. Por esta razón y para no penalizar en exceso la máquina compartida, no debéis dejar mucho tiempo ejecutando en el servidor la práctica a máxima velocidad.

El programa debe estar preparado para que, si el usuario pulsa las teclas CTRL+C desde el terminal, la ejecución del programa termine en ese momento y adecuadamente. Ni en una terminación como esta, ni en una normal, deben quedar procesos en ejecución ni mecanismos IPC sin haber sido borrados del sistema. Este es un aspecto muy importante y se penalizará bastante si la práctica no lo cumple.

Es probable que necesitéis semáforos o buzones para sincronizar adecuadamente la práctica. En ningún caso podréis usar en vuestras prácticas más de un array de semáforos, un buzón de paso de

mensajes y una zona de memoria compartida. Se declarará un array de semáforos de tamaño adecuado a vuestros requerimientos, el primero de los cuales se reservará para el funcionamiento interno de la biblioteca. El resto, podéis usarlos libremente.

La biblioteca requiere memoria compartida. Debéis declarar una única zona de memoria compartida en vuestro programa. Los 2048 bytes primeros de dicha zona estarán reservados para la biblioteca. Si necesitáis memoria compartida, reservad más cantidad y usadla a partir del byte bimilésimo cuadragésimo noveno.

Las funciones proporcionadas por la biblioteca `libbatracios.a` son las que a continuación aparecen. De no indicarse nada, las funciones devuelven -1 en caso de error o, si siendo funciones booleanas, el resultado es falso. Las funciones devuelven 0 en caso contrario:

- `int BATR_inicio(int ret, int semAforos, int lTroncos[],int lAguas[],int dirs[], int tCriar, char *zona)`
  El primer proceso, después de haber creado los mecanismos IPC que se necesiten y antes de haber tenido ningún hijo, debe llamar a esta función, indicando en `ret` la velocidad de presentación y en `tCriar` el tiempo medio entre partos de una rana madre (parámetros ambos de la línea de órdenes) y pasando además el identificador del conjunto de semáforos que se usará y el puntero a la zona de memoria compartida declarada para que la biblioteca pueda usarlos. El significado del resto de parámetros es el siguiente:
  - `lTroncos`: array de siete enteros que contiene el valor de la longitud media de los troncos para cada fila. El índice cero del array se refiere a la fila superior de troncos.
  - `lAguas`: igual que el parámetro anterior, pero referido a la longitud media del espacio entre troncos.
  - `dirs`: lo mismo que los dos parámetros anteriores, pero en esta ocasión cada elemento puede valer `DERECHA`(0) o `IZQUIERDA`(1), indicando la dirección en que se moverán los troncos.
  - Los valores del array de troncos y aguas deben ser todos estrictamente mayor que cero.
- `int BATR_avance_troncos(int fila)`
  Hace avanzar una posición la fila de troncos en su dirección
- `void BATR_descansar_criar(void)`
  Cada rana madre llama a esta función antes de dar a luz a una nueva ranita
- `int BATR_parto_ranas(int i,int *dx,int *dy)`
  La rana madre llama a esta función para tener una ranita. El primer parámetro es el número de rana madre (de 0, rana de la izquierda a 3, rana de la derecha). En `dx` y `dy`, se devuelve la posición donde nace la rana. La rana madre, debe crear un nuevo proceso (o esperarse si ahora mismo hay el máximo) para que se haga cargo de la nueva rana
- `int BATR_puedo_saltar(int x,int y,int direcciOn)`
  Una rana situada en `x` e `y` pregunta con esta función si puede avanzar en la dirección (`DERECHA`, `IZQUIERDA` o `ARRIBA`). Devuelve 0 si la rana puede saltar
- `int BATR_explotar(int x,int y)`
  (*A partir de la versión 0.3*). Hace que la rana situada en la posición que se le pasa, explote y desaparezca. La rana se suma a la cuenta de ranas perdidas. Se usa en la versión en que no se mueven los troncos para matar a las ranas que se quedan atrapadas
- `int BATR_avance_rana_ini(int x,int y)`, `int BATR_avance_rana(int *x,int *y,int direcciOn)` y `int BATR_avance_rana_fin(int x,int y)` Una vez la rana sabe que puede avanzar, llama a estas tres funciones. Los parámetros son de significado evidente. No obstante, fijaos en que la segunda función recibe la posición pasada por referencia, de modo que, una vez realizado el avance, las nuevas coordenadas aparecen en las variables pasadas. Esas mismas nuevas coordenadas, se pasan a la última función
- `int BATR_pausa(void)` e `int BATR_pausita(void)`
  Estas dos funciones producen una pausa, sin consumo de CPU
- `int BATR_comprobar_estadIsticas(int r_nacidas, int r_salvadas, int r_perdidas)`
  Antes de que la práctica termine, hay que comprobar que la suma de ranas nacidas tiene que ser igual al número de ranas salvadas más el número de ranas perdidas más el número de ranas que hay en la pantalla. Esta función realiza la comprobación y se le ha de pasar la cuenta que hayáis realizado vosotros

- `int BATR_fin(void)`
  El padre, una vez sabe que ha acabado la práctica y antes de realizar limpieza de procesos y mecanismos IPC debe llamar a esta función.

Notas acerca de las funciones:

1. Se puede establecer la longitud media de troncos y su separación de todas las filas salvo la del medio. Independientemente del valor espedificado, esta fila presenta una sucesión de troncos de tamaño dos separados también una distancia de dos caracteres.
2. La secuencia para que una rana se mueva consiste en llamar primero a `BATR_avance_rana_ini`, luego a `BATR_avance_rana`, ambas con la posición donde se encuentra la rana. Al retornar de `BATR_avance_rana`, en las variables de posición ya se encontrará la nueva posición de la rana. Hay que llamar a la función `BATR_pausa` y, finalmente, a `BATR_avance_rana_fin`, con la nueva posición de la rana.

Estad atentos pues pueden ir saliendo versiones nuevas de la biblioteca para corregir errores o dotarla de nuevas funciones.

El guión que seguirá el proceso padre será el siguiente:

1. Tomará los datos de la línea de órdenes y los verificará.
2. Iniciará las variables, mecanismos IPC, manejadoras de señales y demás.
3. Llamará a la función `BATR_inicio`.
4. Creará los procesos que se harán cargo de las ranas madre.
5. Entrará en un bucle infinito del que solamente saldrá si se pulsa CTRL+C. En ese bucle, se encargará de mover una a una cada fila de troncos. Entre fila y fila, hará una llamada a la función BATR_pausita
6. Cuando se pulse CTRL+C, se engargará de finalizar todo ordenadamente y comprobará las estadísticas con la función `BATR_comprobar_estadIsticas`

Por su parte, los procesos encargados de manejar a las ranas madre, también están en un bucle infinito:

1. Llama.a la función `BATR_descansar_criar`
2. Si el número de procesos que hay es el máximo, tiene que esperar, sin consumo de CPU, a que haya un "hueco" para el nuevo proceso.
3. Tiene una rana, llamando a `BATR_parto_ranas` y crea un nuevo proceso para que se encargue de la rana recién nacida

Finalmente, los procesos de las ranitas en su bucle infinito, hacen lo que se ha indicado más arriba hasta que desaparecen de la pantalla, bien por la parte de arriba (salvadas) o por un lateral (perdidas)

Observad que existe mucha sincronización que no se ha declarado explícitamente y debéis descubrir dónde y cómo realizarla. Os desaconsejamos el uso de señales para sincronizar. Una pista para saber dónde puede ser necesaria una sincronización son frases del estilo: "después de ocurrido esto, ha de pasar aquello" o "una vez todos los procesos han hecho tal cosa, se procede a tal otra".

Respecto a la sincronización interna de la biblioteca, se usa el semáforo reservado para conseguir atomicidad en la actualización de la pantalla y las verificaciones. Para que las sincronizaciones que de seguro deberéis hacer en vuestro código estén en sintonía con las de la biblioteca, debéis saber que sólo las funciones que actualizan valores sobre la pantalla están sincronizadas mediante el semáforo de la biblioteca.

En esta práctica no se podrán usar ficheros para nada, salvo que se indique expresamente. Las comunicaciones de PIDs o similares entre procesos, si hicieran falta, se harán mediante *mecanismos IPC*.

Siempre que en el enunciado o LPEs se diga que se puede usar `sleep()`, se refiere a la *llamada al sistema*, no a la orden de la línea de órdenes.

Los mecanismos IPC (semáforos, memoria compartida y paso de mensajes) son recursos muy limitados. Es por ello, que vuestra práctica sólo podrá usar un conjunto de semáforos, un buzón de paso de mensajes y una zona de memoria compartida como máximo. Además, si se produce cualquier error o se finaliza normalmente, los recursos creados han de ser eliminados. Una manera fácil de lograrlo es registrar la señal SIGINT para que lo haga y mandársela uno mismo si se produce un error.

### Biblioteca de funciones `libbatracios.a`

Con esta práctica se trata de que aprendáis a sincronizar y comunicar procesos en UNIX. Su objetivo no es la programación, aunque es inevitable que tengáis que programar. Es por ello que se os suministra una biblioteca estática de funciones ya programadas para tratar de que no debáis preocuparos por la presentación por pantalla, la gestión de estructuras de datos (colas, pilas, ...) , etc. También servirá para que se detecten de un modo automático errores que se produzcan en vuestro código. Para que vuestro programa funcione, necesitáis la propia biblioteca `libbatracios.a` y el fichero de cabeceras `batracios.h`. La biblioteca funciona con los códigos de VT100/xterm, por lo que debéis adecuar vuestros simuladores a este terminal. También se usa la codificación UTF-8, por lo que necesitáis un programa de terminal que sepa interpretarlos. Los terminales de Linux lo hacen por defecto, pero si usáis Windows, debéis aseguraros de que el programa tiene capacidad para interpretarlos y que esta capacidad está activada. Si no es así notaréis caracteres basura en la salida de modo que no se verá nada. Es, además, conveniente que pongáis el color de fondo de la pantalla a negro y su tamaño, al menos, a 80x25 caracteres.

#### Ficheros necesarios:

- `libbatracios.a`: para Solaris (ver 0.3), para el LINUX de clase (ver 0.3),
- `batracios.h`: Para todos (ver 0.3).

#### Registro de versiones:

- 0.1: primera versión
- 0.2: corrección de una deriva en la coordenada x
- 0.3:
    1. mejora en las animaciones
    2. pausa mayor en los errores, mostrando el lugar del error
    3. Nueva función: BATR_explotar

## 2. Pasos recomendados para la realización de la práctica

Aunque ya deberíais ser capaces de abordar la práctica sin ayuda, aquí van unas guías generales:

1. Crear los semáforos y la memoria compartida, y comprobad que se crean bien, con `ipcs`. Es preferible, para que no haya interferencias, que los definái privados.
2. Registrad SIGINT para que cuando se pulse CTRL+C se eliminen los recursos IPC. Lograr que si el programa acaba normalmente o se produce cualquier error, también se eliminen los recursos (mandad una señal SIGINT en esos casos al proceso padre).
3. Tratad los parámetros de la línea de órdenes
4. Llamar a la función `BATR_inicio` en `main`. Debe aparecer la pantalla de bienvenida y, pasados dos segundos, dibujarse la pantalla. Añadid también la función `BATR_fin`
5. Probad a que el padre mueva los troncos en la pantalla
6. Dejad para las siguientes pruebas, los troncos quietos
7. Cread los procesos de las ranas madre. Observad cómo se produce un error porque la segunda cría de una madre aparece encima de la anterior, porque no se mueve

8. Haced que las ranitas nacidas se muevan, en principio solamente hacia adelante
9. Sincronizad mediante un semáforo que las madres no den a luz si el cupo de procesos está completo
10. Sincronizad también que una madre no dé a luz hasta que su hija recién nacida haya dejado la posición inicial
11. Las ranas deben evitar ahora avanzar si no pueden hacerlo. Pueden intentar avanzar en otras direcciones y, de no poder, hacer una pausa
12. El padre debe comprobar ahora las estadísticas
13. Finalmente, los troncos deben moverse. Pero ¡cuidado!, las ranitas deben enterarse de cuando han derivado en el río cuando traten de moverse otra vez. Si no, la biblioteca no encontrará a la rana en la posición que dais. Esta es una parte complicada.
14. Activad todo finalmente.
15. Pulid los últimos detalles.
16. Si la cosa marcha, celebradlo como vuestra edad os lo permita. Os lo merecéis. Después de un largo y duro recorrido, sabe como ninguna otra cosa el llegar a la meta.

## 3. Plazo de presentación.

Consultad la página de entrada a la asignatura.

## 4. Normas de presentación.

Acá están. Además de estas normas, en esta práctica se debe entregar un esquema donde aparezcan los semáforos usados, sus valores iniciales, sus buzones, y mesajes pasados y un seudocódigo sencillo para cada proceso con las operaciones *wait* y *signal*, *send* y *receive* realizadas sobre ellos. Por ejemplo, si se tratara de sincronizar dos procesos C y V para que produjeran alternativamente consonantes y vocales, comenzando por una consonante, deberíais entregar algo parecido a esto:

```
SEMáFOROS Y VALOR INICIAL: SC=1, SV=0.

SEUDOCÃ“DIGO:

        C                                V
       ===                              ===
  Por_siempre_jamás               Por _siempre_jamás
     {                                {
      W(SC)                            W(SV)
      escribir_consonante              escribir_vocal
      S(SV)                            S(SC)
      }                                }
```

Daos cuenta que lo que importa en el pseudocódigo es la sincronización. El resto puede ir muy esquemático. Un buen esquema os facilitará muchísimo la defensa.

## 5. Evaluación de la práctica.

Dada la dificultad para la corrección de programación en paralelo, el criterio que se seguirá para la evaluación de la práctica será: si

a. la práctica cumple las especificaciones de este enunciado y,
b. la práctica no falla en ninguna de las ejecuciones a las que se somete y,
c. no se descubre en la práctica ningún fallo de construcción que pudiera hacerla fallar, por muy remota que sea esa posibilidad...

se aplicará el principio de "presunción de inocencia" y la práctica estará aprobada. La nota, a partir de ahí, dependerá de la simplicidad de las técnicas de sincronización usadas, la corrección en el tratamiento de errores, la cantidad y calidad del trabajo realizado, etc.

Debido a la complejidad de la práctica, se permitirá presentarla sin que los troncos se muevan. En ese caso, no obstante, la nota máxima que se puede obtener será de seis puntos. Cuando los troncos no se mueven, algunas ranas pueden quedar atrapadas. Se debe usar en este caso la función `BATR_explotar` para eliminar esa rana. En ese caso, la cuenta de ranas perdidas, se incrementa.

## 6. LPEs.

I. ¿Se puede usar la biblioteca en un Linux de 64 bits? Aquí se os indican las claves.
II. ¿Se puede proporcionar la biblioteca para el Sistema Operativo X, procesador Y? Por problemas de eficiencia en la gestión y mantenimiento del código no se proporcionará la biblioteca más que para Solaris-SPARC y Linux-Intel de 32 bits. A veces podéis lograr encontrar una solución mediante el uso de máquinas virtuales.
III. ¿Dónde poner un semáforo? Dondequiera que uséis la frase, "el proceso puede llegar a esperar hasta que..." es un buen candidato a que aparezca una operación *wait* sobre un semáforo. Tenéis que plantearos a continuación qué proceso hará *signal* sobre ese presunto semáforo, dónde lo hará y cuál será el valor inicial.
IV. Si ejecutáis la práctica en *segundo plano* (con ampersand (&)) es normal que al pulsar CTRL+C el programa no reaccione. El terminal sólo manda `SIGINT` a los procesos que estén en primer plano. Para probarlo, mandad el proceso a primer plano con `fg %` y pulsad entonces CTRL+C.
V. Un "truco" para que sea menos penoso el tratamiento de errores consiste en dar valor inicial a los identificadores de los recursos IPC igual a -1. Por ejemplo, `int semAforo=-1`. En la manejadora de `SIGINT`, sólo si `semAforo` vale distinto de -1, eliminái s el recurso con `semctl`. Esto es lógico: si vale -1 es porque no se ha creado todavía o porque al intentar crearlo la llamada al sistema devolvió error. En ambos casos, no hay que eliminar el recurso.
VI. Para evitar que todos los identificadores de recursos tengan que ser variables globales para que los vea la manejadora de `SIGINT`, podéis declarar una estructura que los contenga a todos y así sólo gastáis un identificador del espacio de nombres globales.
VII. A muchos os da el error "Interrupted System Call". Mirad la sesión dedicada a las señales, apartado quinto. Allí se explica lo que pasa con `wait`. A vosotros os pasa con `semop`, pero es lo mismo. De las dos soluciones que propone el apartado, debéis usar la segunda.
VIII. A muchos, la práctica os funciona exasperantemente lenta en encina. Debéis considerar que la máquina cuando la probáis está cargada, por lo que debe ir más lento que en casa o en el linux de clase.
IX. A aquellos que os dé "Bus error (Core dumped)" al dar valor inicial al semáforo, considerad que hay que usar la versión de `semctl` de Solaris (con `union semun`), como se explica en la sesión de semáforos y no la de HPUX.
X. Al acabar la práctica, con CTRL+C, al ir a borrar los recursos IPC, puede ser que os ponga "`Invalid argument`", pero, sin embargo, se borren bien. La razón de esto es que habéis registrado la manejadora de `SIGINT` para todos los procesos. Al pulsar CTRL+C, la señal la reciben todos, el padre y los otros procesos. El primero que obtiene la CPU salta a su manejadora y borra los recursos. Cuando saltan los demás, intentan borrarlos, pero como ya están borrados, os da el error.
XI. El compilador de encina tiene un bug. El error típicamente os va a ocurrir cuando definái s una variable entera en memoria compartida. Os va a dar `Bus Error. Core dumped` si no definís el puntero a esa variable apuntando a una dirección que sea múltiplo de cuatro. El puntero que os devuelve `shmat`, no obstante, siempre será una dirección múltiplo de cuatro, por lo que solo os tenéis que preocupar con que la dirección sea múltiplo de cuatro respecto al origen de la memoria compartida. La razón se escapa un poco al nivel de este curso y tiene que ver con el alineamiento de direcciones de memoria en las instrucciones de acceso de palabras en el procesador RISC de encina.
XII. Se os recuerda que, si ponéis señales para sincronizar esta práctica, la nota bajará. Usad semáforos, que son mejores para este cometido.
XIII. Todos vosotros, tarde o temprano, os encontráis con un error que no tiene explicación: un proceso que desaparece, un semáforo que parece no funcionar, etc. La actitud en este caso no es tratar de justificar la imposibilidad del error. Así no lo encontraréis. Tenéis que ser muy

sistemáticos. Hay un árbol entero de posibilidades de error y no tenéis que descartar ninguna de antemano, sino ir podando ese árbol. Tenéis que encontrar a los procesos responsables y tratar de localizar la línea donde se produce el error. Si el error es "`Segmentation fault. Core dumped`", la línea os la dará si aplicáis lo que aparece en la sección Manejo del depurador. En cualquier otro caso, no os quedará más remedio que depurar mediante órdenes de impresión dentro del código.

Para ello, insertad líneas del tipo:

```
fprintf(stderr,"...",...);
```

donde sospechéis que hay problemas. En esas líneas identificad siempre al proceso que imprime el mensaje. Comprobad todas las hipótesis, hasta las más evidentes. Cuando ejecutéis la práctica, redirigid el canal de errores a un fichero con `2>salida`.

Si cada proceso pone un identificador de tipo "P1", "P2", etc. en sus mensajes, podéis quedaros con las líneas que contienen esos caracteres con:

```
grep "P1" salida > salida2
```

XIV. Os puede dar un error que diga `Resource temporarily unavailable` en el `fork` del padre. Esto ocurre cuando no exorcizáis adecuadamente a los procesos hijos zombies del padre. Hay dos posibilidades para solucionarlo:
   1. La más sencilla es hacer que el padre ignore la señal `SIGCLD` con un `sigaction` y `SIG_IGN`. El S.O. tradicionalmente interpreta esto como que no queréis que los hijos se queden zombies, por lo que no tenéis que hacer waits sobre ninguno de ellos para que acaben de morir
   2. Interceptar SIGCLD con una manejadora en el padre y, dentro de ella, hacer los waits que sean necesarios para que los hijos mueran. Pero esto trae un segundo problema algo sutil: al recibirse la señal, todos los procesos bloqueados en cualquier llamada al sistema bloqueante (en particular, los WAITs de los semáforos) van a fallar. Si no habéis puesto comprobación de errores, los semáforos os fallarán sin motivo aparente. Si la habéis puesto, os pondrá `Interrupted system call` en el perror. Como podéis suponer, eso no es un error y debéis interceptarlo para que no ponga el perror y reintente el WAIT. La clave está en la variable `errno` que valdrá `EINTR` en esos casos.

XV. No se debe dormir (es decir, ejecutar `sleeps` o `pausas`) dentro de una sección crítica. El efecto que se nota es que, aunque la práctica no falla, parece como si solamente un proceso se moviera o apareciera en la pantalla a la vez. Siendo más precisos, si dormís dentro de la sección crítica, y soltáis el semáforo para, acto seguido, volverlo a coger, dais muy pocas posibilidades al resto de procesos de que puedan acceder.

XVI. La biblioteca no reintenta los WAITs de su semáforo, por lo que, de recibirse una señal, podría fallar. Si os da problemas, simplemente ignorad la señal SIGCLD en el padre como se explica más arriba.

XVII. Si la práctica se para al llamar el padre al final a la función `BATR_comprobar_estadIsticas`, puede ser por lo siguiente: casi todas las funciones de la biblioteca están sincronizadas mediante el semáforo que se pasa a la biblioteca. Ese semáforo se usa para construir una sección crítica de modo que la salida por la pantalla no se embarulle. Ocurre que, cuando se recibe SIGINT, puede que a algún proceso, sobre todo a máxima velocidad, le pille dentro de una función de la biblioteca con el semáforo cogido. El proceso morirá y el semáforo se queda a cero. Cuando el padre llama a la función para comprobar las estadísticas, intenta hacer un wait sobre el semáforo y se queda bloqueado para siempre. La solución pasa por, o bien bloquear SIGINT en los sitios conflictivos o bien que el padre, una vez se asegure de que todos los demás procesos están muertos, compruebe el valor del semáforo de la biblioteca y, si vale cero, hacerle un signal.

XVIII. A algunos de vosotros se os produce un curioso error cuando contáis las ranas. Usáis memoria compartida para que los procesos puedan contar las ranas y, cuando la cuenta vale 127 y se incrementa, pasa a valer -128. El problema radica en que estáis usando punteros de tipo `char` para las cuentas. Un `char` se desborda al llegar a 127. Debéis usar punteros de tipo `int` y reservar cuatro bytes (mejor sería `sizeof(int)` bytes) de memoria compartida para cada uno de ellos.